CSE394: Security Policy Frameworks
Scott Stoller
Project. Version: 18oct2005a.

An initial proposal for your course project is due in class on October 27. Some ideas appear below. This is not a comprehensive list. Feel free to suggest other topics. Of course, you are welcome to discuss ideas with me before submitting the initial proposal.

You may work in teams. Each team should submit one proposal, with the names of all team members on it. The size of the team should be proportional to the size of the proposed project.

I will provide feedback on the proposals (on the topic and the size of the project) within a week of submission.

Depending on the number of teams and the project topics, I might ask some or all teams to present their projects in class, during the last two weeks of class. This lets teams benefit from each other's experience, and it can provide good practice in public speaking.

Printouts of final versions of all documents (i.e., revised versions of all previously submitted documents, and all remaining documents) are due on Friday, December 9. Electronic submission of all project-related files (documents and code) is due by midnight on the same day. The files should be submitted (in one .zip or .tgz archive) by email to `cse394@cs.sunysb.edu`.

You may use any available software in your implementation, provided you indicate the source.

1 Security Policy Development

Develop prototypical security policies for some application domain. You will probably need to do some research on the selected application domain, unless you already have experience with it. When you are unable to find details about some aspect of actual policies in the application domain, you can invent something plausible for that aspect of your policy (better, list some plausible alternatives you considered and then choose one). *The informal description of your policy must clearly indicate which parts of your policy are based on other resources, and which parts you invented.*

Policy development projects are generally suitable for teams with 1 or 2 members, depending on the scope and level of detail of the policy.

You are welcome to propose any application domain, and I will help you determine whether it is appropriate. Here are some suggestions.

- University. Universities have detailed policies and procedures restricting access to information about students (including academic and financial information) by faculty, staff, and external organizations. These policies are based in part on state and federal laws, such as FERPA. There are also policies governing access to information about faculty and staff, especially information (such as the standardized teaching evaluations by students, and letters of recommendation from students, faculty in other institutions, and faculty at Stony Brook) used in tenure and promotion decisions. I posted a file with some information about SBU's privacy policy on the CSE394 Announcements page.

  These policies sometimes require that certain accesses are allowed if appropriate approvals have been granted. For example, a student can specifically authorize release of his or her transcript to a specified entity. As another example, the author of a letter of recommendation can specify whether the letter is accessible to the person being evaluated, and under what conditions (e.g., he can read it but not make copies, or he can read only an abbreviated version with the author's name removed). Note that the security policy should specify these conditions even if they cannot be enforced automatically by the computer system.

Although the university is a single institution, the policy and its administration are decentralized within the university's organizational structure (the main levels are university, campus, college, and department), and there are interactions with external entities, such as high schools, other universities (e.g., students transferring to or from Stony Brook), students' prospective employers, the state government, accreditation agencies (e.g., ABET), sports organizations (e.g., NCAA can check that football players are taking the required number of credits and have the required GPA), etc.

- Health care. My suggestion here is to extend Becker's EHR policy with policies for other organizations. Becker specified policies for the Spine, local health organizations (hospitals and doctor's offices), the nationwide registration authority, and the nationwide patient demographic service. Other related organizations include insurance companies, billing and accounting companies (doctor's offices and hospitals often outsource these tasks), and government agencies (such as Medicare and the FDA). You might need to adapt Becker's policy to fit the American health care system.
- Financial institution. To keep the project manageable, you should focus on some line of business, such as commercial banking or equities trading, but still consider policies governing interactions between that line of business and other lines of business and other entities (customers, regulatory agencies, corporate partners, advertising agencies, etc.).
- Government agency. For example, a police department, the IRS, the FDA, etc. The policy should consider access to organizational information (i.e., who is allowed to appoint employees to positions, assign them to projects/tasks, set their schedules, etc.) and agency-specific information (e.g., for a police department, information about criminals, on-going investigations, etc.). It should also consider interactions between the agency and external entities (e.g., for a police department, interactions with other police departments, the judicial system, the FBI, etc.).

The deliverables are:

Proposal: The proposal should identify the team members, indicate the application domain, sketch the aspects you will consider, and list some resources (books, web sites, people, etc.) that you plan to use for information about the application domain.

AppDomain: Introduction to the application domain, including typical system architectures, and typical organizational and administrative structures. Be sure to indicate the relationship between the system architecture and the administrative structure: who administers each part of the system?

InformalPolicy: An informal (English) description of the security policy. "Informal" is not a synonym for "vague." Be precise. Organize the policy and your presentation of it, by dividing it into sections. Security policy administration is very important and is considered part of the security policy. The informal policy should be free from "implementation bias". In other words, your development of the desired policy should not be influenced by premature concerns about how it will be expressed in a particular policy framework or how it will be implemented. The policy should be accompanied by explanation and justification, to help readers understand it.

FormalPolicy: A formal specification of the security policy in some policy framework, accompanied by an explanation of any non-obvious aspects of the translation from the informal policy to the formal policy. The policy framework may be role-based access control (based on the ANSI standard, or some other source) or trust management (such as Cassandra). Feel free to modify or extend the framework as appropriate for your application domain; just be sure to explain your modifications and extensions explicitly and clearly.

Implementation: An implementation of the security policy and some testcases. The goal of the implementation is to support evaluation of your policy ("is X allowed? what about Y?"), not to provide a production system. Therefore, the implementation can be centralized, even if actual deployment would require a distributed implementation. Similarly, use of real digital signatures and other authentication mechanisms is not required.

UserManual: A brief user manual, describing how to install and run your system and some illustrative testcases.

Your prototype, although centralized, should simulate trust negotiation. For example, suppose the policy contains the rule `A@A.r(args) :- B@C.r1(args1), A@A.r2(args2)`. We can see from the conclusion that this rule is part of `A`'s policy. Premises with location `A` (e.g., the premise `A.r2(args2)`) can be evaluated directly, but premises with a location different than `A` (e.g., the premise `B@C.r1(args1)`), may be evaluated only after checking whether the request (for the credential) is allowed. In Cassandra, this check is done by the Access Control Engine, which implements the Cassandra API. We want to avoid implementing that API in the prototype. So, we integrate the check into the policy as an additional premise of the rule. Continuing the example, we would replace the above rule with `A.r(args) :- B.canRequestCredential(A, B.r1(args1)), B.r1(args1), A.r2(args2)`.

Your policy rules should specify appropriate locations to obtain credentials, but the locations can be ignored in your centralized implementation of your policy.

For formal policies expressed as rules, two implementation strategies are:

- Use a constraint logic-programming (CLP) system. Two recommended CLP systems are mentioned below. You do not need to implement an API for your system; you can let the user exercise your policy by directly entering queries in the CLP system. The advantage of this strategy is that the implementation will look very much like the formal policy: you just need to type in the formal policy, with some syntactic changes. The disadvantage is that you will need to learn to use a CLP system (hey, this sounds more like an advantage).

- Use an object-oriented language, such as Java. The implementation should have the same structure as the formal policy, to the greatest extent possible, so it is easy to (1) verify that the the formal policy is implemented correctly, and (2) update the implementation to reflect changes to the formal policy. Efficiency is a secondary concern. Each code fragment corresponding to a policy rule should be preceded by a comment that contains the rule or a reference to it. Before you start coding, you should think carefully about how to translate the rule-based policy into the object-oriented language, and your user manual should describe your translation strategy. (For example, there is an object corresponding to each issuer, and it has a method corresponding to each relation for which it issues facts, and ...)

As indicated below, you have 3 weeks to write AppDomain and InformalPolicy. This is the hardest part of the project. Don't leave it for the third week.

2 Security Policy Infrastructure

My main suggestion in this category is to design and implement a system that supports trust management, with a rule-based policy language similar to Cassandra.

The supported policy language does not need to be exactly the same as Cassandra, but you should describe and justify the differences. For example, you can change the syntax to be closer to Prolog, e.g., permit(issuer,entity,action) instead of issuer.permit(entity,action). Since aggregation in Cassandra is used mostly to express negation, you can eliminate most uses of aggregation (which is not a standard feature of Prolog) and use Prolog's negation operator directly. I didn't check systematically, but the only other use of aggegation that I noticed in Becker's EHR policy is the use of aggregation in rules S2.2.17 and S5.3.1 to check whether all relevant third parties have given consent for the patient to read an item in the patient's EHR; you can implement this using the standard Prolog predicate findall (it is documented in Section 6.9 of volume 1 of the XSB manual).

Implementation of the Cassandra API, following the pseudo-code in the trust management tutorial slides that we used in class, should be relatively straightforward. You do not need to support credentials that contain constraints. The main issue is how to implement the policy evaluator. You could implement your own policy evaluator (in Java, C++, or whatever), or you can interface to an existing constraint logic programming (CLP) system. I think the latter will be easier; there is no point in re-implementing the wheel. If you really want to implement your own policy evaluator, it should include some basic optimizations (e.g., use a hash map to find relevant rules and facts quickly, and cache derived ground facts).

Available CLP systems do not support credential gathering, but you can implement credential gathering in external functions (implemented in Java or C++). When the CLP system reaches a premise with a remote location, if the desired fact is not already in the policy (it could have been added to the policy as a result of a previous request for it), the system should request a credential. You should replace such premises with external functions that request the credential. If the requested credential is received, the external function adds the received fact to the policy (i.e., to the policy loaded in the CLP process, not the policy stored on disk; this caches the fact for possible re-use) and returns true; otherwise, the external function returns false. A natural approach is to use an external function whose name is based on the name of the relation in the premise, but it will be easier to use a single external function for all premises with remote locations. This external function accepts the name of the relation as an argument. For example, suppose the policy contains the rule A@A.r(args) :- B@C.r1(args1), A@A.r2(args2). The local premise can be evaluated directly. The remote premise can be satisfied by a locally cached credential or by a call to an external function that communicates with B. The former case is handled by the existing rule. The latter case is handled by adding a second rule in which the remote premise is replaced with, e.g., eval_remote_premise(B,C,r1,args1).

There is a remaining problem, related to the fact that the available CLP systems are (as far as I know) single-threaded. Suppose the trust management system at location A requests a credential C1 from location B; then the former is blocked in the external function, waiting for B's reply. Suppose B, on receiving this request, requests credential C2 from A before replying; this could easily happen during trust negotiation. A's instance of the CLP system is blocked on the request for C1, so it cannot handle B's request. The simplest solution is for the trust management system at A to start another process running the CLP system and use it to respond to the request for C2. Each new CLP process should be initialized by loading the local policy and cached credentials. Starting a CLP process is somewhat expensive, so the new CLP process should be kept in case it is needed again. More generally, the trust management system should maintain a set of CLP processes, keep track of their status (i.e., whether each one is available or waiting for a reply to a request), and create new ones as needed. When a credential is received, it should be given to all of the CLP processes as soon as possible.

You do not need to use secure communication. You do not need to use digital signatures on certificates; you can assume that no one tries to cheat. Secure communication and signatures could easily be added later.

To demonstrate your system, you should implement all or most of Becker's EHR policy in it. Your documentation should clearly indicate which (if any) parts of the EHR policy are unimplemented or have been modified. You can convert the PDF file to plain text, so you don't need to type all of the rules from scratch; you just need to adapt the syntax to your system. I converted it using Adobe Acrobat and posted the resulting text file on the CSE394 Announcements page. In Becker's EHR policy, EHR data is external to the policy, and is accessed via external functions. You can keep it external, and store it in a DBMS (accessed directly from the CLP process, or accessed via the host process using JDBC or ODBC), or, for simplicity, you can move the EHR database into the policy (i.e., store EHR data as facts in the policy).

This project is suitable for teams with 2 or 3 members, with two differences. (1) Teams with two members do not need to support multiple CLP processes; in other words, it is acceptable if their system deadlocks during trust negotiation scenarios like the one described above. (2) Teams with three members should store EHR data in a DBMS (about half of the students in this class have taken a course on databases, and a few students are taking one now, so this requirement should not be problematic). Teams with 2 members can get extra credit by supporting multiple CLP processes or storing EHR data in a DBMS.

Another possible project is to extend the J2EE security model, which supports a simple form of role-based access control, to support a richer policy language, for example, some form of trust management, and implement the extended model in an open-source J2EE framework, such as JBoss, available at www.jboss.org. JBoss is big; just identifying the relevant parts could be a challenge.

Deliverables for a security policy infrastructure project are:

Design: A description of the design of your system.

Implementation: The implementation. The code should be well documented.

UserManual: A user manual, including instructions for installing, compiling, and running your system.

SampleApp: A sample application, with documentation of how to run it.

3 Schedule

The following table specifies due dates for deliverables of both kinds of projects. Documents should be submitted as printouts in class except where the table specifies otherwise. "Contributions" is a document that briefly describes the contributions of each team member to the project since the previous submission. Status reports for infrastructure projects should indicate which parts of the system and sample application have been implemented, and which parts have been tested; also, if there have been changes to the design since the previous submission, please describe the changes in the status report. The user manual due on Nov 24 is not expected to be the final version but should be mostly finished.

Date	**Policy Devel.**	**Infrastructure**
Oct 25	Proposal (approx. 1 page)	
Nov 15	AppDomain, InformalPolicy, Contributions	Status Report, Design, Contributions
Nov 24	FormalPolicy, Contributions	Status Report, User Manual, Contributions
Dec 9	Printout of all documents, including another Contributions, due at 2pm	
Dec 9	Electronic submission of everything due at midnight	
Dec 12-15	Demos	

4 Other Projects

If you are interested in working on any other topic, you are welcome to discuss the topic with me, during office hours or by appointment, and submit a project proposal for it. The topic must be related to Security Policy Frameworks, but it does not need to be related specifically to material covered in class. Include a list of deliverables, with scheduled delivery dates, in the proposal.

5 Grading

We will first grade the projects themselves, and then determine each team member's grade for the project based on that member's contributions to the project, taking the Contributions reports and all other available information into account.

6 Constraint Logic Programming Systems

If you use a CLP system, I suggest that you consider XSB or Eclipse. If you find other CLP systems that look promising, please let me know.

XSB is available at xsb.sourceforge.net. XSB has a foreign language interface for calling C and C++ functions. XSB supports an interface, called InterProlog (the XSB home page contains a link to it), that allows host applications written in Java to send queries to Eclipse processes, and allows XSB processes to invoke methods in the Java host process. XSB has a database interface based on ODBC. XSB supports Constraint Handling Rules (CHR), which is a mechanism for extending logic programming systems to handle constraints. CHR is designed to make CLP systems open-ended: adding support for a constraint domain ideally just requires writing a few high-level constraint-handling rules, which, roughly speaking, are re-write rules that describe how to simplify those constraints. XSB is a mature and powerful system, developed mainly at Stony Brook. It uses tabling for efficient top-down evaluation. Support for CHR has been added relatively recently, largely by Tom Schrijvers at Katholieke Universiteit Leuven in Belgium. For more on CHR in XSB, see volume 2 of the XSB manual, available from the above URL. For even more details, see the publications at www.cs.kuleuven.ac.be/~toms/Research/CHR/.

Eclipse (not the Java IDE) is available at www.icparc.ic.ac.uk/eclipse/. Eclipse has a foreign language interface for calling C and C++ functions. It also has an interface that allows host applications written in Java to communicate with Eclipse processes. Eclipse has existing support for numerous (more numerous than XSB) constraint domains, some built-in and some implemented using Constraint Handling Rules. It also has support for suspending goals; this feature could be used to avoid creating multiple CLP processes during trust negotiation.

With either system, the standard Prolog predicates `assert` and `retract` (and their variants) can be used to add and remove facts (e.g., `hasActivated` and `isDeactivated` facts, and facts received in credentials) from the loaded policy in the CLP process. They are documented in section 6.13 of volume 1 of the XSB manual. We are not concerned with fine-tuning the performance of the CLP system, so you can ignore the comments about indexing and optimization.